Project 1:

Title:
Mastermind

Class:
CSC 7 - 44265

Due Date:
April 3rd, 2022

Author:
Aamir Khan

Introduction and Gameplay:

Title: Mastermind

Objective:

Mastermind is a code-breaking game that's usually played between two players. The objective of Mastermind is to break a code or correctly guess a random code that is either randomly generated or chosen by the other player. This code can be either 4, 6, or 8 digits long. If a player can manage to break the code generated within a specific number of attempts, the player wins. Otherwise, the game ends and no winner is declared. To make the game more difficult for players, the option of using duplicate digits in the random code generated is available. This complicates the game because now more options can be tested or guessed as a result of allowing duplicates to be used.

The "How-To":

Each round consists of 10 attempts that the current player has to correctly guess or break the code. After each attempt, hints will be displayed to assist the player in breaking the random generated code. An 'R' indicates that a digit is correct and in the correct place. However, the player won't know exactly what digit is correct or in the correct place. A 'W' indicates that a digit is correct but in the incorrect place. This means that 1 of the digits is correct, but that it needs to be placed in a different position. Finally, a '_' indicates that a digit is incorrect and in the wrong place. This will tell the player that the digit doesn't exist in the code to break, so the player can then eliminate that particular digit from their guesses moving forward.

Overview:

For this project, I utilized dynamic memory allocation to create my game objects. I also utilized functions, 2-D arrays, input validation, pointers (Both 1-dimensional and 2-dimensional), looping constructs, conditional constructs, ternary operators, and procedural programming. I could add onto this project by creating a Mastermind class, but I choose not to for this project to demonstrate my understanding of procedural programming. The problem I struggled with the most in this project was getting the logic of the game down. For example, deciding what hints to show for a given guess was very challenging because of the fact that duplicates can be used in the random generated code. I had to tweak my functions multiple times throughout my versions as I wanted to make sure the program would run smoothly regardless of whether we use duplicates or not. To my benefit, this program wasn't too long or time consuming to code.

Stages of Development:

Version 1: Creating Board with 1 Column

For the very first version of my project, I went ahead and created a 2-dimensional char array that would represent the game board. To start things off on a simple note, I randomly generate a 1 digit code to break and ask the user/player to guess the code within 10 tries. Since the code-to-break and all guesses are 1 digit in length, I only need 1 column in my 2-dimensional char array to hold all of the guesses that are made.

Version 2: Creating Board with 2 Columns

For this version of my project, I get the code to work with a code length of 2. In other words, the code to break and all guesses need to be two digits in length. As such, I need char arrays for both the code to break and the guesses that will be made. I use input validation to ensure that the guesses that are made are the correct length and consist of digits between 0 and 7, inclusive. I use ternary operators to output the winning/losing message as well as how many tries were used.

Version 3: Implementing Functions

In this version, I created and implemented functions. In doing so, I managed to make main look cleaner and utilize functions to get everything done within fewer lines of code. I made the process of creating a board into a function as well as the process of making guesses into functions. I also made a function to recover and deallocate all of the memory dynamically allocated within my program. A few more functions were created to simplify other processes.

Version 4: Increasing Code Length to 4

I don't add much to this program besides getting the program to finally work with a code length of 4. In the previous versions, the game wasn't exactly realistic because the code length would need to be either 4, 6, or 8.

Version 5: No Duplicates Allowed

For this version of my project, I introduce the first option available to users, using no duplicates. To do this, I use input validation on the code to break to ensure that the code generated is unique. I also use input validation on my guesses to make sure they don't contain duplicate digits. Although guesses can't contain duplicate digits in this version, I fix this by Version 10. Essentially, even though no duplicates are used in the code to break that is generated, the player should be able to type in any values including duplicate digits. Specifying no duplicates to be used should only specify how we want our code to break to be generated, not how our guesses should be structured.

Version 6: Showing Hints with No Duplicates

In this version of my project, I finally get the hints working with codes that are unique. As mentioned in “The How-To” section, An ‘R’ indicates that a digit is correct and in the correct place. A ‘W’ indicates that a digit is correct but in the incorrect place. Finally, a ‘_’ indicates that a digit is incorrect and in the wrong place. These hints are displayed after each guess is made.

Version 7: Allowing Duplicates

In this version, I introduce the second option available to players, allowing duplicates to be used. As we should expect, the code that is randomly generated can now contain duplicate digits, thus making the game more difficult to win. In most of the previous versions including this one, the code to break is visible as I had to test whether my program was functioning properly. I had to make sure that the game’s logic was working as it should with my programming logic.

Version 8: Showing Hints with Duplicates & No Duplicates

For this version of my project, I go through and get the hints working for both options. Showing the hints when no duplicates were used was simple enough, but there’s more thought that has to go into getting the appropriate hints when duplicates are used. As long as a hint wasn’t already set, I went through and set the appropriate hint. However, if there was already a hint in place, I didn’t want to overwrite the hint. The improved function made sure that no duplicate hints were displayed and that the hints were correct.

Version 9: Cleaning up Main

In this version, I simplified main even more and got the game to work within just a few lines of code in main. I popped the entire game into its own function. That way, I could play an entire game of Mastermind with just one function call.

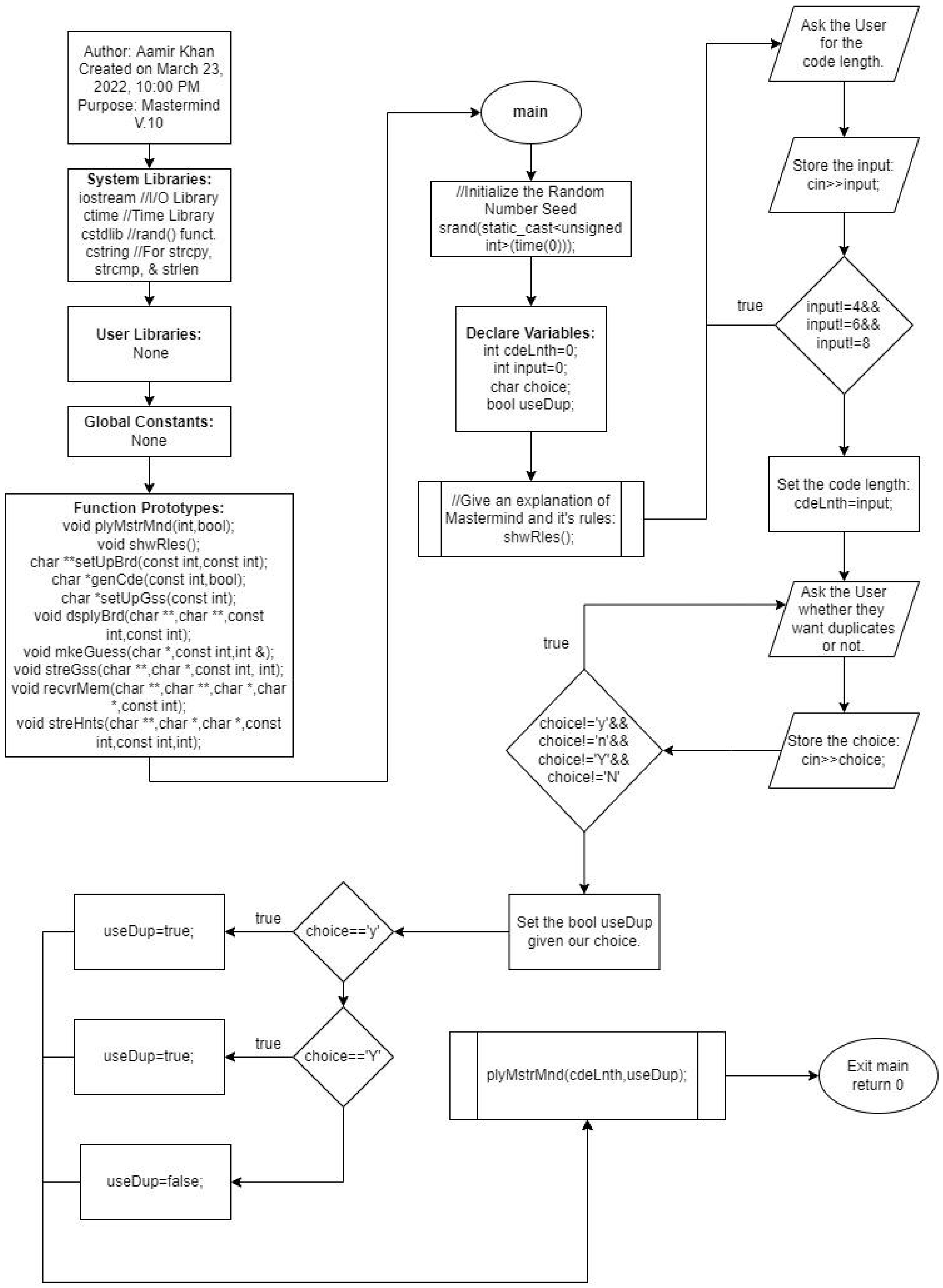
Version 10: Finishing Touches

I added some finishing touches to my project and touched up some functions. I made sure that the player could input any values, including duplicates, regardless of whether duplicates were allowed or not in the code to break. The player should be able to type in any values in their guess; only the code to break needs to be validated given the user’s option to allow or not allow duplicates. In short, not allowing duplicates simply ensures that the code to break is unique, it doesn't mean duplicates can’t be used in the player’s guess.

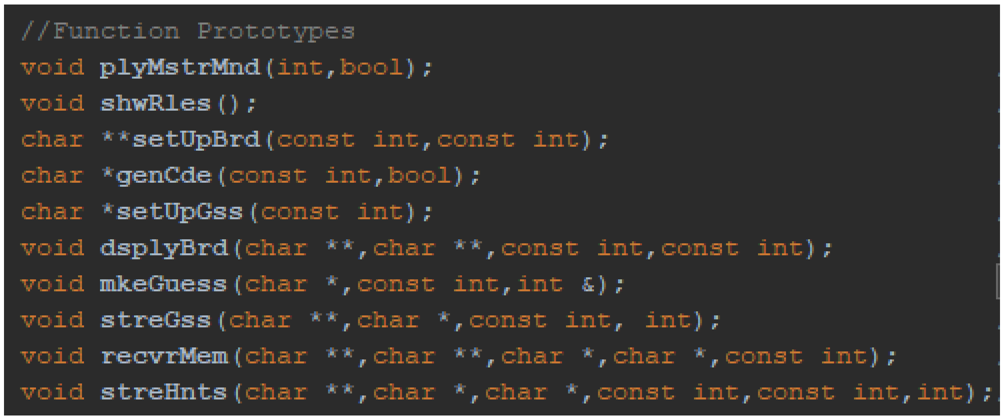
Mastermind Pseudo-Code:

```
/*
 * File:   main.cpp
 * Author: Aamir Khan
 * Created on March 21, 2022, 9:30 AM
 * Purpose: Mastermind Pseudo-Code
 */

//System Libraries
//I/O Library
//Cstd. Library
//Ctime Library
//Cstring Library
//Namespace Std of System Libraries

//User Libraries

//Global Constants

//Function Prototypes
//List my function prototypes.

//Main -> Execution Begins Here:

        //Initialize the Random # Generator Seed.

        //Declare My Variables and Initialize Them.

        //Display the Rules and Description of Mastermind.

        //Ask the User for the Code Length They Want, Validate It, and Store the Input.

        //Ask the User Whether They Want to Use Duplicates or Not, Validate It, and Store It.

        //Play a game of Mastermind.

        //Quit the Program.

//Exit main - End of the Program
```

Mastermind Flowchart:

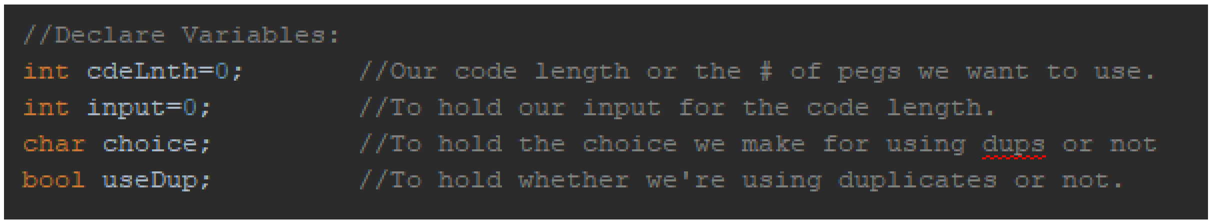

Author: Aamir Khan
Created on March 23, 2022, 10:00 PM
Purpose: Mastermind V.10
System Libraries:
iostream //I/O Library
ctime //Time Library
cstdlib //rand() funct.
cstring //For strcpy, strcmp, & strlen
User Libraries:
None
Global Constants:
None
Function Prototypes:
void plyMstrMnd(int,bool);
void shwRles();
char **setUpBrd(const int,const int);
char *genCde(const int,bool);
char *setUpGss(const int);
void dsplyBrd(char **,char **,const int,const int);
void mkeGuess(char *,const int,int &);
void streGss(char **,char *,const int, int);
void recvrMem(char **,char **,char *,char *,const int);
void streHnts(char **,char *,char *,const int,const int,int);
main
//Initialize the Random Number Seed
srand(static_cast<unsigned int>(time(0)));
Declare Variables:
int cdeLnth=0;
int input=0;
char choice;
bool useDup;
//Give an explanation of Mastermind and it's rules:
shwRles();
Ask the User for the code length.
Store the input:
cin>>input;
true
input!=4&&
input!=6&&
input!=8
Set the code length:
cdeLnth=input;
Ask the User whether they want duplicates or not.
true
Store the choice:
cin>>choice;
choice!='y'&&
choice!='n'&&
choice!='Y'&&
choice!='N'
Set the bool useDup given our choice.
true
choice=='y'
useDup=true;
true
choice=='Y'
useDup=true;
useDup=false;
plyMstrMnd(cdeLnth,useDup);
Exit main
return 0

Sample Screen Shots:



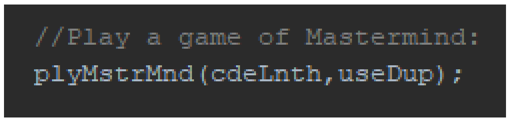
```
//Function Prototypes
void plyMstrMnd(int,bool);
void shwRles();
char **setUpBrd(const int,const int);
char *genCde(const int,bool);
char *setUpGss(const int);
void dsplyBrd(char **,char **,const int,const int);
void mkeGuess(char *,const int,int &);
void streGss(char **,char *,const int, int);
void recvrMem(char **,char **,char *,char *,const int);
void streHnts(char **,char *,char *,const int,const int,int);
```

(A list of all function prototypes. These prototypes can be found in Version 10.)



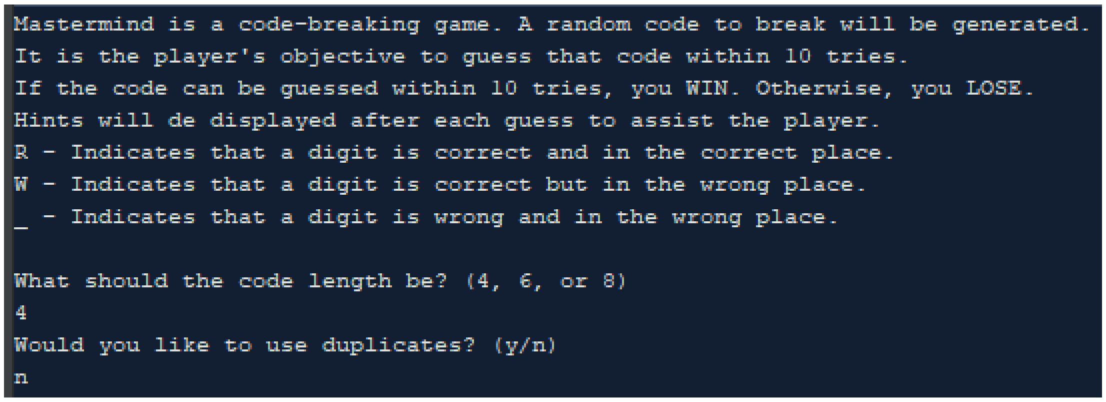
```
//Declare Variables:
int cdeLnth=0;          //Our code length or the # of pegs we want to use.
int input=0;            //To hold our input for the code length.
char choice;            //To hold the choice we make for using dups or not
bool useDup;            //To hold whether we're using duplicates or not.
```

(The variables found in main. Code length and useDup are reinitialized later on given our input.)



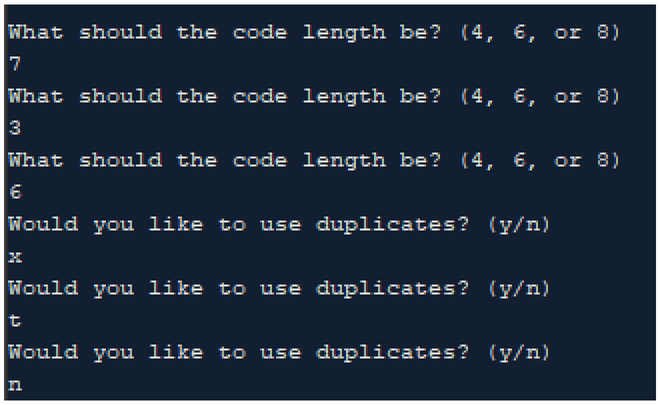
```
//Play a game of Mastermind:
plyMstrMnd(cdeLnth,useDup);
```

A one-line function call that lets users play a single game of Mastermind (Only 2 inputs necessary).



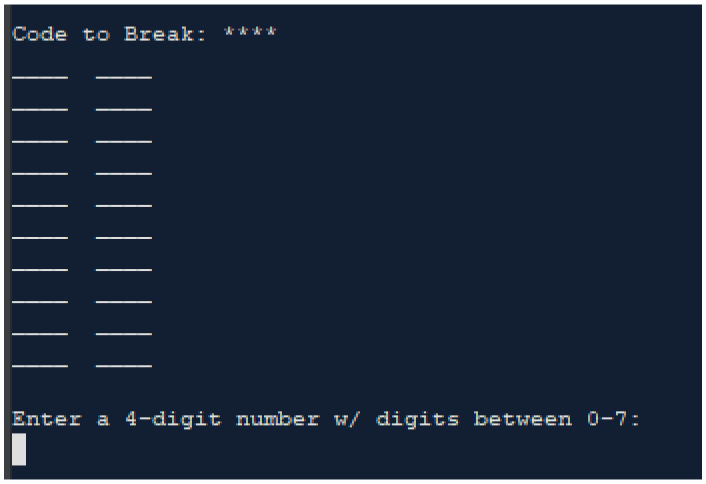
```
Mastermind is a code-breaking game. A random code to break will be generated.
It is the player's objective to guess that code within 10 tries.
If the code can be guessed within 10 tries, you WIN. Otherwise, you LOSE.
Hints will de displayed after each guess to assist the player.
R - Indicates that a digit is correct and in the correct place.
W - Indicates that a digit is correct but in the wrong place.
_ - Indicates that a digit is wrong and in the wrong place.

What should the code length be? (4, 6, or 8)
4
Would you like to use duplicates? (y/n)
n
```

(Displaying the rules and a brief description of Mastermind. We also display some prompts.)



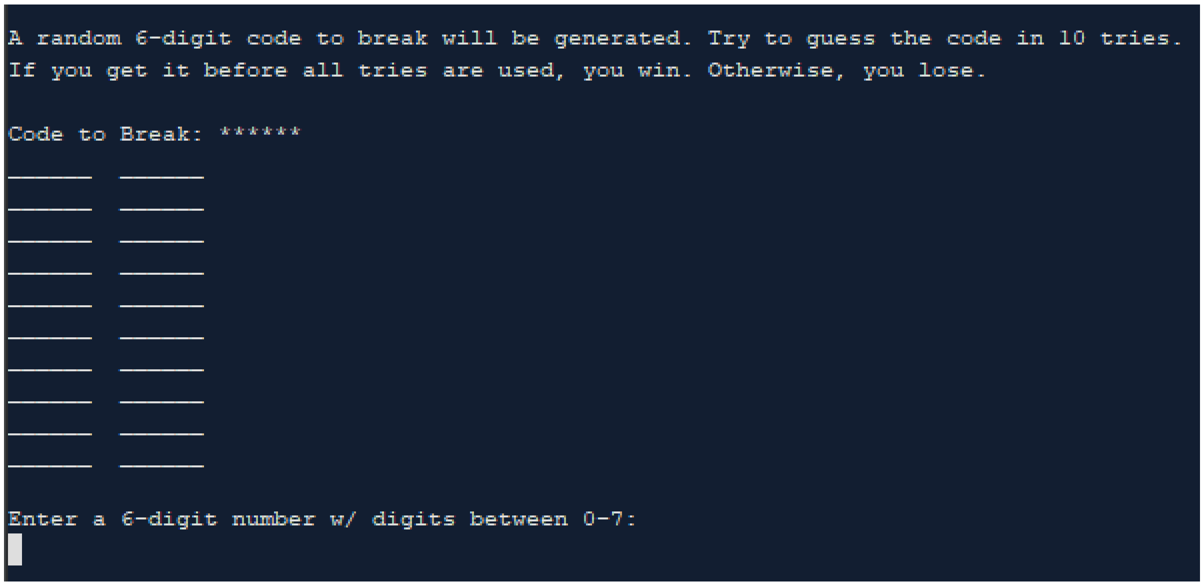
```
What should the code length be? (4, 6, or 8)
7
What should the code length be? (4, 6, or 8)
3
What should the code length be? (4, 6, or 8)
6
Would you like to use duplicates? (y/n)
x
Would you like to use duplicates? (y/n)
t
Would you like to use duplicates? (y/n)
n
```

A look at some input validation. We prompt the user for the code length they'd like for the game, and we reprompt as long as the input isn't any of the 3 available options. We also prompt the user for whether they'd like to use duplicates in the generated code. If their input isn't a y/n, we keep prompting the user for a valid input/choice.



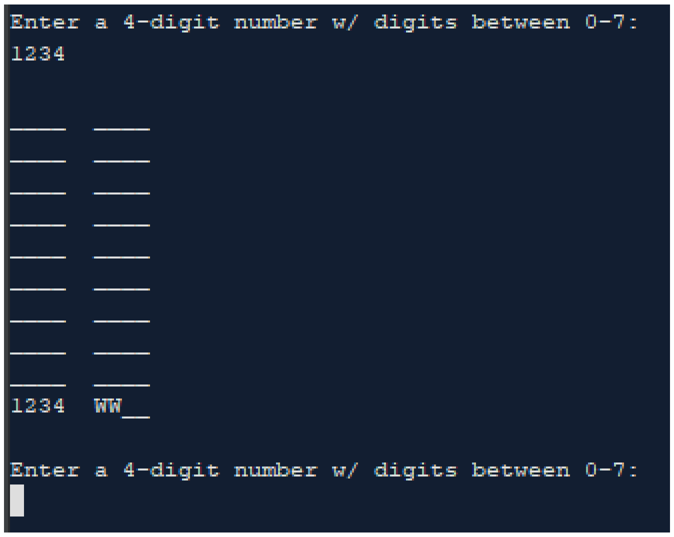
```
Code to Break: ****
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____

Enter a 4-digit number w/ digits between 0-7:
```

This image shows how the layout of our guess board looks, and right beside it we have our hint board. Every guess we make will be stored on a row starting from the bottom. To the right of each guess will be a hint for that particular guess. We store both guesses and hints for these particular guesses to provide a way for players to see exactly what they've already guessed and see all of their hints for these guesses.



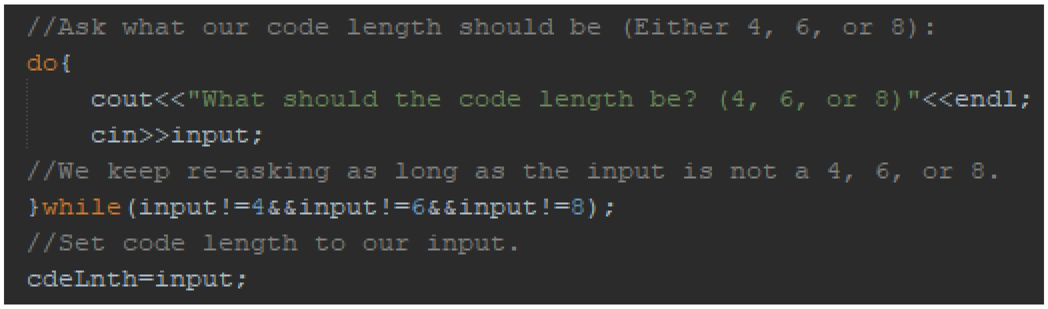
```
A random 6-digit code to break will be generated. Try to guess the code in 10 tries.
If you get it before all tries are used, you win. Otherwise, you lose.

Code to Break: ******
______  ______
______  ______
______  ______
______  ______
______  ______
______  ______
______  ______
______  ______
______  ______
______  ______

Enter a 6-digit number w/ digits between 0-7:
```

(This image showcases an example of starting the round with a code length besides 4.)



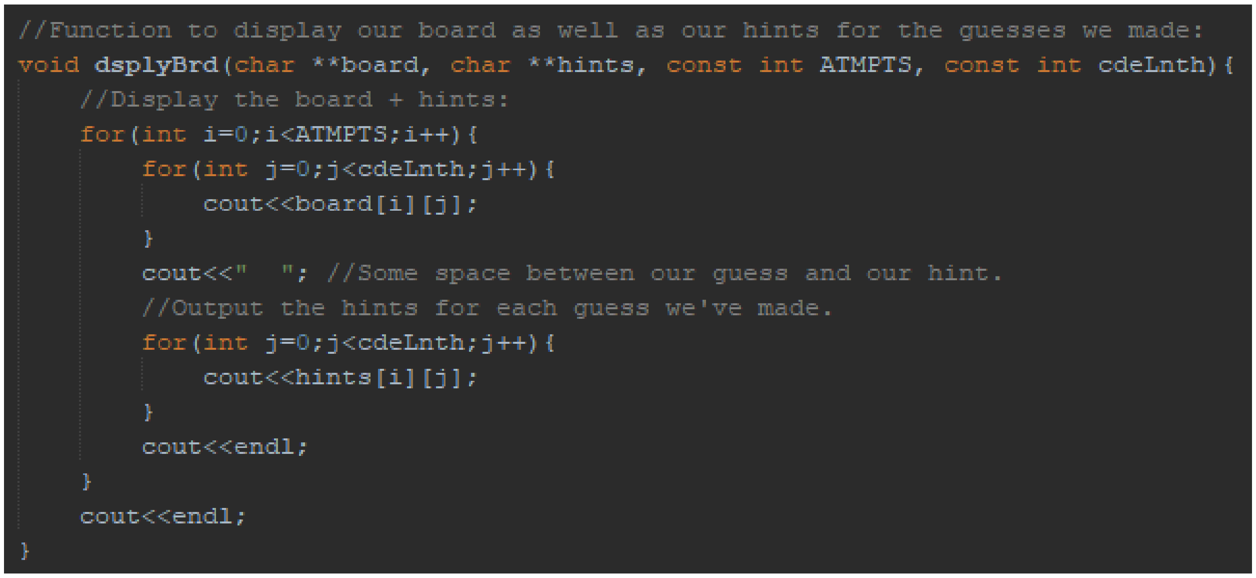
```
Enter a 4-digit number w/ digits between 0-7:
1234

____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
____  ____
1234  WW__

Enter a 4-digit number w/ digits between 0-7:
```

This image demonstrates how guesses are stored and how hints for these guesses are stored. Guesses and hints are stored from bottom to top and will be displayed each time we display our board. In this example, the hint generated has two W's which indicate that 2 of our digits in our guess are correct but that they're in the wrong place. Moreover, we don't know exactly which digits are correct; we simply know how many are correct in our guess.



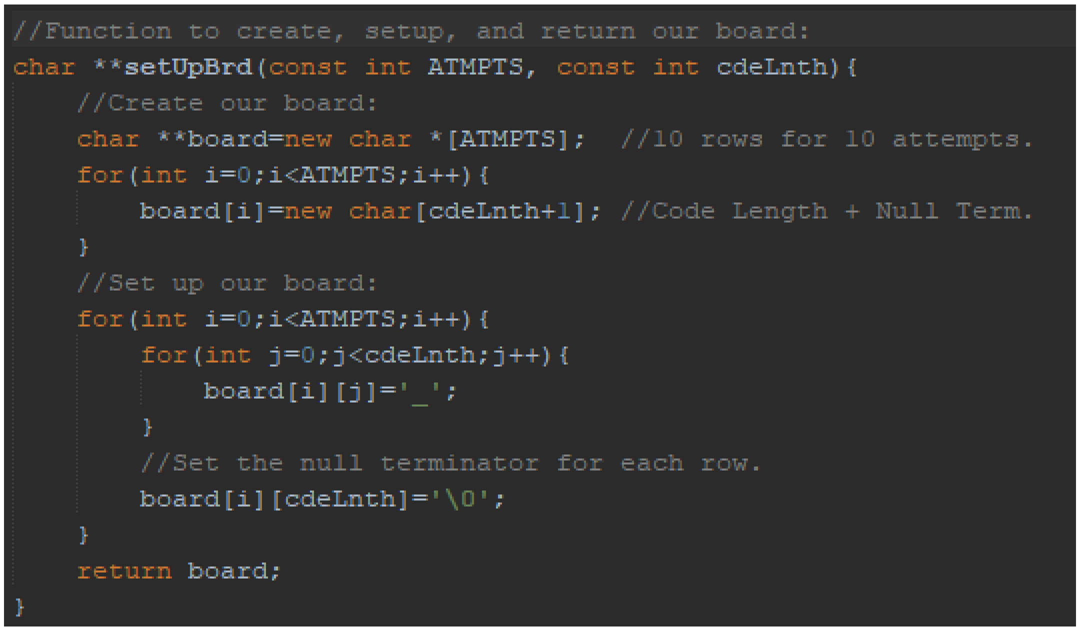
```
//Ask what our code length should be (Either 4, 6, or 8):
do{
    cout<<"What should the code length be? (4, 6, or 8)"<<endl;
    cin>>input;
//We keep re-asking as long as the input is not a 4, 6, or 8.
}while(input!=4&&input!=6&&input!=8);
//Set code length to our input.
cdeLnth=input;
```

(This is an example of how I validated the input for the code length using a do-while.)



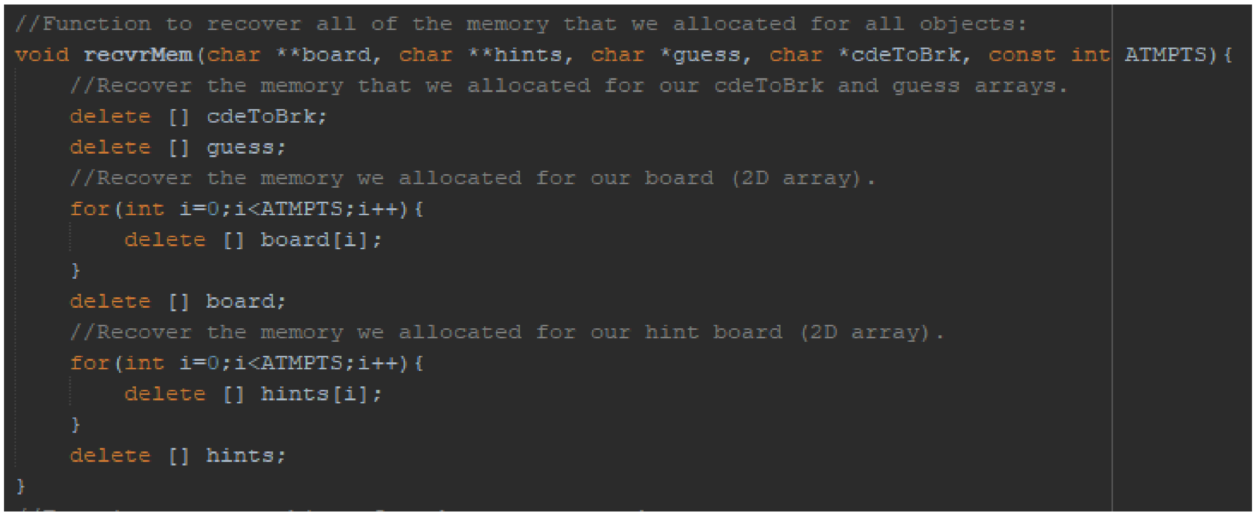
```
//Function to display our board as well as our hints for the guesses we made:
void dsplyBrd(char **board, char **hints, const int ATMPTS, const int cdeLnth){
    //Display the board + hints:
    for(int i=0;i<ATMPTS;i++){
        for(int j=0;j<cdeLnth;j++){
            cout<<board[i][j];
        }
        cout<<"  "; //Some space between our guess and our hint.
        //Output the hints for each guess we've made.
        for(int j=0;j<cdeLnth;j++){
            cout<<hints[i][j];
        }
        cout<<endl;
    }
    cout<<endl;
}
```

(We use an outer for loop for the rows and inner for loops to correspond with our columns. The first inner loop is used to display our guess for the given row (All column elements for a given row). The second inner for loop is used to display our hints for the guess on the given row.)



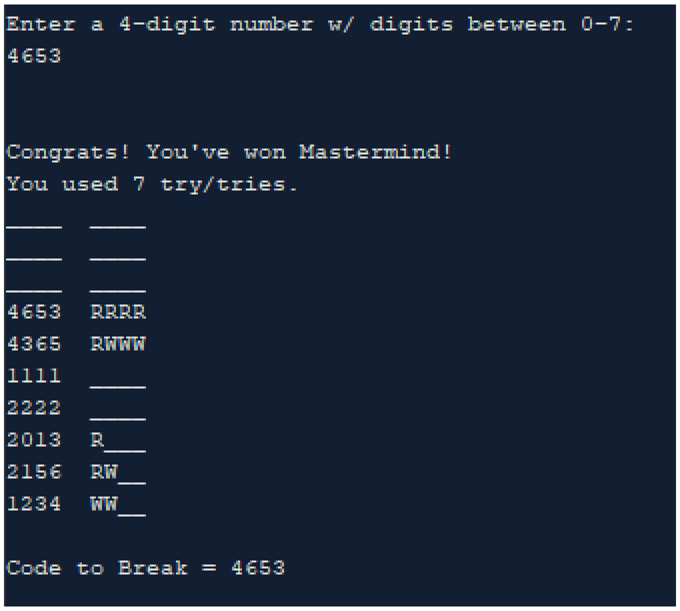
```
//Function to create, setup, and return our board:
char **setUpBrd(const int ATMPTS, const int cdeLnth){
    //Create our board:
    char **board=new char *[ATMPTS];  //10 rows for 10 attempts.
    for(int i=0;i<ATMPTS;i++){
        board[i]=new char[cdeLnth+1]; //Code Length + Null Term.
    }
    //Set up our board:
    for(int i=0;i<ATMPTS;i++){
        for(int j=0;j<cdeLnth;j++){
            board[i][j]='_';
        }
        //Set the null terminator for each row.
        board[i][cdeLnth]='\0';
    }
    return board;
}
```

(We utilize a 2-dimensional character array for our board as well as our hints that correspond to our board. We dynamically allocate memory for a 2D character array and return it.)



```
//Function to recover all of the memory that we allocated for all objects:
void recvrMem(char **board, char **hints, char *guess, char *cdeToBrk, const int ATMPTS){
    //Recover the memory that we allocated for our cdeToBrk and guess arrays.
    delete [] cdeToBrk;
    delete [] guess;
    //Recover the memory we allocated for our board (2D array).
    for(int i=0;i<ATMPTS;i++){
        delete [] board[i];
    }
    delete [] board;
    //Recover the memory we allocated for our hint board (2D array).
    for(int i=0;i<ATMPTS;i++){
        delete [] hints[i];
    }
    delete [] hints;
}
```

(As I've dynamically allocated memory throughout my program, I need to deallocate the memory that I've used for my board, hints, guess, and code to break. We do this before exiting main.)



```
Enter a 4-digit number w/ digits between 0-7:
4653


Congrats! You've won Mastermind!
You used 7 try/tries.
____  ____
____  ____
____  ____
4653  RRRR
4365  RWWW
1111  ____
2222  ____
2013  R___
2156  RW__
1234  WW__

Code to Break = 4653
```

This is a test run of my program in which I've specified a code length of 4 and have chosen not to use duplicates. Notice that I'm able to type in any values, including duplicates. Choosing to not use duplicates simply ensures that the code to break is unique, it doesn't mean I can't make guesses with duplicates. I managed to win the game within 7 tries, so a winning message was displayed. I also output the code that was to be broken. That way, if we had lost, we'd be able to see what the correct code to break was.